Research Report 301

Dynamic Scheduling in the Presence of Faults: Specification and Verification

Tomasz Janowski, Mathai Joseph

RR301

A distributed real-time program is usually executed on a limited set of hardware resources and is required to satisfy timing constraints, despite anticipated hardware failures. Static analysis of the timing properties of such programs is often infeasible. This paper shows how to formally reason about these programs when scheduling decisions are made on-line and take into account deadlines, load and hardware failures. We use Timed CCS as a process description language, define a language to describe anticipated faults and apply a version of μ-calculus to specify and verify timing properties. This allows the property of schedulability to be the outcome of an equation-solving problem. And unlike conventional reasoning, the logis is *fault-monotonic*: if correctness is proved for a number of faults, correctness for any subset of these faults is guaranteed.

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

March 1996

Dynamic Scheduling in the Presence of Faults: Specification and Verification *

Tomasz Janowski[1] and Mathai Joseph[2]

[1] The United Nations University
International Institute for Software Technology
P.O. Box 3058, Macau
[2] Department of Computer Science
University of Warwick, Coventry CV4 7AL, UK

Abstract. A distributed real-time program is usually executed on a limited set of hardware resources and is required to satisfy timing constraints, despite anticipated hardware failures. Static analysis of the timing properties of such programs is often infeasible. This paper shows how to formally reason about these programs when scheduling decisions are made on-line and take into account deadlines, load and hardware failures. We use Timed CCS as a process description language, define a language to describe anticipated faults and apply a version of a μ-calculus to specify and verify timing properties. This allows the property of schedulability to be the outcome of an equation-solving problem. And unlike conventional reasoning, the logic is *fault-monotonic*: if correctness is proved for a number of faults, correctness for any subset of these faults is guaranteed.

1 Introduction

Consider a real-time system which consists of a fixed number of tasks, each with a possibly unbounded number of invocations. Some tasks are *periodic* and will be invoked at regular intervals by timers; the others are *sporadic* tasks and are invoked by some other task or by the environment. Let the tasks be statically partitioned between the nodes of the system, all connected by a multiple-access network and each providing resources like clocks, memories and processors. Clocks are used to implement timers, and asynchronous communication takes place using memory to implement Protected Shared Objects (PSO's) [5]. There will usually be more tasks than processors, so at each node the allocation of local resources is controlled by a real-time scheduler; there is also a protocol for scheduling the network traffic. The hardware of the system may be unreliable: for example, processors may fail, memory may be corrupted and communication may be delayed.

A real-time system operates under both resource and timing constraints, for example that a task produces 'correct' output within a specified time. But such

* Supported in part by EPSRC research grant GR/H39499.

deadlines need not be restricted to one task and we let a *transaction* relate the timing of actions in one or more tasks. Assume that tasks communicate through PSO's and that normally they are statically scheduled using the ceiling priority protocol [5]. When hardware faults occur, however, we let dynamic scheduling be used for a more flexible reassignment of resources. Verification of timing properties will require some assumptions: (1) about the speed and the number of processors, (2) about anticipated hardware failures and (3) about the minimum inter-arrival time between external invocations for each sporadic task.

Timed process algebras (e.g. [20, 11, 26]) provide an obvious formal framework for this analysis. But based on the *maximal parallelism* assumption [24], most are unable to represent delays due to resource contention or to model scheduling decisions directly. This gave rise to CCSR [9] which provides synchronous timed actions and asynchronous instantaneous events, the former resolving competition for resources and the latter for synchronization. But CCSR (and most other such formalisms) assume the use of fixed priorities and are thus unsuitable for modelling dynamic scheduling decisions, e.g. to recover from faults. Further, bisimulation-based reasoning is usually insufficient to verify fault-tolerance: it may be possible to provably tolerate a number of faults, yet be unable to provably tolerate only some of them [12]. With unpredictable faults, such *fault-monotonicity* is important but is hard to establish in most branching-time theories. Finally, fault-tolerant scheduling has recently received some attention [3, 22] but in a semantic framework which does not give sufficient insight into how proofs of feasibility of other scheduling problems can be obtained, and with synchronization restricted to simple precedence between tasks.

This paper shows how to realistically analyse the timing properties of communicating systems in the framework of timed process algebras. We use a version of Timed CCS [26] and use timed processes to represent tasks, to model hardware and to describe schedulers. Dynamic scheduling is used (unlike [8, 9, 6]) and priorities are assigned to tasks, and not to individual actions (unlike [8, 9]). This makes it possible to use a single framework for reasoning, abstraction and automatic verification [10] and to relate schedulability to equation-solving [21, 17]. The language is equipped with the usual transitional semantics and provides means of representing the effects of faults: semantically, using additional τ-labelled transitions, and syntactically, by 'faulty' declarations for process constants [12]. For verification, a version of the μ-calculus which follows timed [26] and modal [14] extensions of Hennessy-Milner logic is used; this is also fault-monotonic [12]. The logic allows expression of deadlines, transactions and environment assumptions, and is able to detect deadlocks and verify fault-tolerance.

We start by assuming an unlimited number (maximal parallelism) of fault-free hardware resources (Section 2). Assuming fault-free hardware, Section 3 provides an architectural model for describing and reasoning about systems with limited resources. Section 4 retains the assumption of unlimited resources and introduces reasoning about the effects of hardware failures. For this, the process language is assigned different fault-affected semantics, and the logic refined into its fault-monotonic version. These improvements are combined in Section

5, where it is shown how to reason about timing under resource limitations, in the presence of anticipated hardware faults. Section 6 provides a discussion.

2 Resource-based Systems

We first briefly describe a timed process language (based on TCCS) and a logic (based on a modal μ-calculus) to express and verify timing properties, assuming unlimited, fault-free hardware resources.

2.1 The Language

Let $\mathcal{A}$ be a set of actions consisting of untimed actions ($\mathcal{L}$) and timed actions $\varepsilon(t)$, one for each non-negative real t and representing a delay of t units of time. We shall normally exclude $t = 0$ and write τ instead of $\varepsilon(0)$, and represent synchronization by complementary untimed actions a and $\overline{a}$ ($\overline{\overline{a}} = a$ and $\overline{\varepsilon(t)} =_{def} \varepsilon(t)$). Let $L \subseteq \mathcal{L}$, $f : \mathcal{A} \to \mathcal{A}$ ($f(\varepsilon(t)) = \varepsilon(t)$ and $f(\overline{a}) = \overline{f(a)}$) and $X \in \mathcal{X}$, a set of process constants. Also, let $a \in \mathcal{L}$ and $\alpha \in \mathcal{A}$. There are three syntactic categories: process expressions Pe, declarations Δ and processes P.

$$\begin{array}{lll} Pe & ::= & 0 \mid X \mid \alpha.Pe \mid Pe + Pe \\ \Delta & ::= & [\,] \mid \Delta[X \mathrel{\widehat{=}} Pe] \mid \alpha \odot \Delta \mid \Delta \oplus \Delta \\ P & ::= & 0 \mid \mu X.\Delta \mid \alpha.P \mid P + P \mid P|P \mid P \setminus L \mid P[f] \end{array} \tag{1}$$

Informally, 0 represents deadlock, $\alpha.P$ is process P with prefix α and process $P + Q$ represents alternation. $P|Q$ is used for concurrent composition, $P \setminus L$ for restriction, $P[f]$ for renaming and $\mu X.\Delta$ for the solution X of the recursive equations Δ. The declarations include the empty declaration $[\,]$, $\Delta[X \mathrel{\widehat{=}} Pe]$, to declare X as Pe and other constants as in Δ, $\alpha \odot \Delta$ to prefix α to the right side of all declarations in Δ, and $\Delta \oplus \nabla$ to sum the right sides of the corresponding declarations in Δ and ∇. It is assumed that in $\mu X.\Delta$, X and all constants occurring in Δ are also declared in Δ. We abbreviate $[][X \mathrel{\widehat{=}} Pe][Y \mathrel{\widehat{=}} Qe]$ as $[X \mathrel{\widehat{=}} Pe, Y \mathrel{\widehat{=}} Qe]$ and will often write $[X \mathrel{\widehat{=}} Pe \mid p]$ for all declarations $X \mathrel{\widehat{=}} Pe$ such that predicate p holds. Formally, the semantics of Δ is a partial function $[\![\Delta]\!]$ defining the assignment of process expressions to process constants, as in Table 1, and with $dom(\Delta)$ for the constants declared in Δ.

The semantics of processes is operational, and is defined in Table 2 by structural induction, closely following [26]. The first row of rules applies to all untimed actions plus τ: $\eta \in \mathcal{L} \cup \{\tau\}$. Rows two and three define the passing of time and apply to $t, u > 0$. The rules let $a.P$ idle indefinitely until the environment is ready to synchronize. There is no waiting once synchronization is possible (maximal progress), i.e. no delay for $\tau.P$ while $P|Q$ will idle unless P and Q can synchronize: $\mathcal{S}_t(P)$ includes all actions a of P which are possible within t time units. For example, $\mathcal{S}_t(\alpha.P) = \{\alpha\}$, $\mathcal{S}_t(\tau.P) = \emptyset$, $\mathcal{S}_{t+u}(\varepsilon(t).P) = \mathcal{S}_u(P)$ and $\mathcal{S}_t(P|Q) = \mathcal{S}_t(P) \cup \mathcal{S}_t(Q)$. Time is continuous, and delays do not cause loss of actions (persistency) or result in reaching different states (determinacy). Row four applies to both timed and untimed actions. In particular $[\![\Delta]\!](X)\{\widetilde{\mu Y.\Delta/Y}\}$ is a

process obtained by simultaneous substitution of all constants Y in $\mathcal{X}([\![\Delta]\!](X))$ by their corresponding fixed points $\mu Y.\Delta$. In the sequel we shall apply obvious extensions of the language to describe value-passing and assume the usual translation into the basic language [19].

2.2 The Logic

The logic is a version of the modal μ-calculus which follows the timed extensions of HM logic [26] (meaning, among other things, that we need infinite conjunction) and which for simplicity, like the process language, does not allow for nesting of the fixed point operator [13]. Let ε be an empty sequence and let $\widehat{s}$ denote $s \in \mathcal{A}^*$ with all τ's removed (and delays summed).

$$\begin{array}{l} \widehat{\varepsilon} = \varepsilon \quad \widehat{as} = a\widehat{s} \quad \widehat{\tau s} = \widehat{s} \\ \widehat{\varepsilon(t)as} = \varepsilon(t)a\widehat{s} \quad \widehat{\varepsilon(t)\tau s} = \varepsilon(t)\widehat{s} \quad \widehat{\varepsilon(t)\varepsilon(u)s} = \varepsilon(t+u)\widehat{s} \end{array} \tag{2}$$

The formulas F are built using constants tt and identifiers Z, negation, disjunction which is possibly infinite, the existential modality, and the greatest fixed point, with the other operators derived as usual. As in the process language, the syntax consists of formula expressions Fe, declarations ∇ and formulas F.

$$\begin{array}{lll} Fe & ::= & tt \mid Z \mid \neg Fe \mid \bigvee_{i\in I} Fe \mid \langle\widehat{\alpha}\rangle Fe \\ \nabla & ::= & [Z \mathrel{\widehat{=}} Fe] \mid [Z \mathrel{\widehat{=}} Fe]\nabla \\ F & ::= & tt \mid \nu Z.\nabla \mid \neg F \mid \bigvee_{i\in I} F \mid \langle\widehat{\alpha}\rangle F \end{array}$$

For simplicity we write $\nabla(Z)$ for the formula expression which is assigned to Z by ∇ and as usual assume that identifiers in $\nabla(Z)$ occur within an even number of negations, each also declared in ∇. Then the semantics of F (Fe) is defined relative to an assignment δ of identifiers Z to process-sets and is the set $[\![F]\!]_\delta$ of processes that satisfy M (we write $P \models M$ whenever $P \in [\![M]\!]$). Pointwise inclusion and summation is used to define $[\![\nu Z.\nabla]\!]_\delta$ and we let $[\![\nabla]\!]_{\delta'}$ be the assignment of all $Z \in dom(\nabla)$ to $[\![\nabla(Z)]\!]_{\delta'}$ [13].

$$\begin{array}{l} [\![tt]\!]_\delta =_{def} \mathcal{P} \qquad [\![Z]\!]_\delta =_{def} \delta(Z) \qquad [\![\neg M]\!]_\delta =_{def} \mathcal{P} - [\![M]\!]_\delta \\ [\![\bigwedge_{i\in I} M_i]\!]_\delta =_{def} \bigcap_{i\in I} [\![M_i]\!]_\delta \qquad [\![\nu Z.\nabla]\!]_\delta =_{def} \bigcup\{\delta' \mid \delta' \subseteq [\![\nabla]\!]_{\delta'}\}(Z) \\ [\![[\widehat{\alpha}]M]\!] =_{def} \{P \mid \forall P', t (P \xrightarrow{t} P' \wedge \widehat{t} = \widehat{\alpha}) \Rightarrow P' \in [\![M]\!]_\delta\} \end{array}$$

Table 1. Denotational semantics of declarations.

$dom([\,]) =_{def} \emptyset$		
$[\![\alpha \odot \Delta]\!](X) =_{def} \alpha.[\![\Delta]\!](X)$		if $X \in dom(\Delta)$
$[\![\Delta[Y \mathrel{\widehat{=}} Pe]]\!](X) =_{def}$	Pe	if $X = Y$
	$[\![\Delta]\!](X)$	if $X \neq Y$, $X \in dom(\Delta)$
$[\![\Delta \oplus \nabla]\!](X) =_{def}$	$[\![\Delta]\!](X)$	if $X \in dom(\Delta) - dom(\nabla)$
	$[\![\Delta]\!](X) + [\![\nabla]\!](X)$	if $X \in dom(\Delta) \cap dom(\nabla)$
	$[\![\nabla]\!](X)$	if $X \in dom(\nabla) - dom(\Delta)$

Table 2. Operational semantics of processes.

$$\frac{}{\eta.P \xrightarrow{\eta} P} \quad \frac{P \xrightarrow{\eta} P'}{P+Q \xrightarrow{\eta} P'} \quad \frac{Q \xrightarrow{\eta} Q'}{P+Q \xrightarrow{\eta} Q'} \quad \frac{P \xrightarrow{\eta} P'}{P|Q \xrightarrow{\eta} P'|Q} \quad \frac{Q \xrightarrow{\eta} Q'}{P|Q \xrightarrow{\eta} P|Q'} \quad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

$$\frac{}{0 \xrightarrow{\varepsilon(t)} 0} \quad \frac{}{a.P \xrightarrow{\varepsilon(t)} a.P} \quad \frac{}{\varepsilon(t+u).P \xrightarrow{\varepsilon(t)} \varepsilon(u).P} \quad \frac{}{\varepsilon(u).P \xrightarrow{\varepsilon(u)} P}$$

$$\frac{P \xrightarrow{\varepsilon(t)} P'}{\varepsilon(u).P \xrightarrow{\varepsilon(t+u)} P'} \quad \frac{P \xrightarrow{\varepsilon(t)} P' \quad Q \xrightarrow{\varepsilon(t)} Q'}{P+Q \xrightarrow{\varepsilon(t)} P'+Q'} \quad \frac{P \xrightarrow{\varepsilon(t)} P' \quad Q \xrightarrow{\varepsilon(t)} Q'}{P|Q \xrightarrow{\varepsilon(t)} P'|Q'}, \ \mathcal{S}_t(P) \cap \overline{\mathcal{S}_t(Q)} = \emptyset$$

$$\frac{P \xrightarrow{\alpha} P'}{P \setminus L \xrightarrow{\alpha} P' \setminus L}(\alpha, \overline{\alpha} \notin L) \quad \frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]} \quad \frac{[\![\Delta]\!](X)\{\widetilde{\mu Y.\Delta}/\widetilde{Y}\} \xrightarrow{\alpha} P'}{\mu X.\Delta \xrightarrow{\alpha} P'}(X \in dom(\Delta))$$

Though the logic allows verification of timed processes, in general an unlimited number of processors is assumed to be available to execute concurrent tasks. Consider, for instance, n independent sporadic tasks P_i, each invoked by an action a_i and responding with $\overline{b_i}$ after t_i units of time, perhaps representing the speed of the underlying processor: $P_i =_{def} \mu X.[X \mathrel{\widehat{=}} a_i.\varepsilon(t_i).\overline{b_i}.X]$. Let

$$\begin{aligned} F =_{def} \textstyle\bigwedge_{i=1}^{n} \nu Z(i).\ [Z(i) \quad &\mathrel{\widehat{=}} \textstyle\bigwedge_{\alpha \neq a_i} [\alpha] Z(i) \wedge [a_i] Z'(i,0)] \\ [Z'(i,t) &\mathrel{\widehat{=}} \langle b_i \rangle tt \wedge [b_i] Z(i) \vee \textstyle\bigvee_{\alpha} \langle \alpha \rangle tt \wedge \\ &\quad \textstyle\bigwedge_{c \neq b_i} [c] Z'(i,t) \wedge \bigwedge_{u \leq t, -t} [\varepsilon(u)] Z'(i, t+u)] \end{aligned}$$

F states that if tasks are taken together then each is either ready for invocation or is able to complete within t_i. Since $|_{i=1}^{n} P_i \models F$, unless processing takes no time, each task must be executed on its own processor.

3 Resource-Limited Systems

In order to reason about the timing properties of communicating systems, it is essential to consider the limitations of the underlying hardware. One approach is to constrain the semantics of parallel composition, so that $P|Q \xrightarrow{\mu} P'|Q$ "not always" follows $P \xrightarrow{\mu} P'$, and then verify the properties as usual; another is have the usual semantics and to verify the properties "relative" to the environment constraints [15]. We take yet another approach which leaves the semantics and the logic unchanged but represents resources syntactically. The mapping between *Tasks* and *Resources* is the goal of the *Scheduler*. Given a set L of scheduling events and the *Timing* properties to be established, finding a feasible scheduler (if any) can be represented as the equation-solving problem [21, 17]:

$$(Tasks|Resources|Scheduler)\backslash L \models Timing$$

This makes it possible to represent limitations in the number and also the speed of processors, so that tasks need not represent delays explicitly. We first show how a scheduler maps tasks to shared resources in a centralized system. We show later how to use a distributed model and how to specify and verify transactions.

3.1 Tasks and Resources

Consider a set of tasks $Task_i$, some of them periodic ($i \in [1, per]$) and invoked by timers, and the others sporadic ($i \in [per+1, per+spo]$), and invoked by the environment or by some other task. Let $Task_i$ be a simple sporadic task which is invoked by action in_i from the environment and which returns a result by out_i. To represent resource-limited executions, let $Task_i$ request a processor ($\overline{req_i}$) immediately after it is invoked and release the processor ($\overline{rel_i}$) when returning the result. To take account of the execution speed, assume that after being allocated, $Task_i$ can only proceed if provided with the actions $tick_i$.

$$Task_i =_{def} in_i(x).\overline{req_i}.\\ \mu X(x).(tick_i \odot \Delta)[Y(y) \mathrel{\widehat{=}} \overline{rel_i}.\overline{out_i(y)}.in_i(x).\overline{req_i}.X(x)]$$

We use $tick_i$ to represent the basic machine cycle of the underlying processor and given $Processor_k$ ($k \in [1, pro]$), $speed_k$ is the minimum time that must elapse between two ticks. Which task is currently executed by $Processor_k$ depends on the value received by the last action prt_k (for pre-empt). The action is available at any time and can pre-empt execution of the current task.

$$Processor_k =_{def} prt_k(i).\\ \mu X(i).[X(i) \mathrel{\widehat{=}} prt_k(j).X(j) + \varepsilon(speed_k).\overline{tick_i}.X(i)]$$

The identity of the executed task is available to $Processor_k$ but the converse is not true; a task may be allocated to different processors during one invocation.

As common in scheduling theory, we assume that tasks cannot voluntarily suspend themselves. One more assumption is that all processors share a common instruction set, each taking a basic machine cycle. Without communication, the declarations $tick_i \odot \Delta$ for independent tasks $Task_i$ can only take two forms:

$$X(x) \mathrel{\widehat{=}} tick_i.X'(f(x))\\ X(x) \mathrel{\widehat{=}} tick_i.\text{if } p(x) \text{ then } X'(x) \text{ else } X''(x)$$

$f(x)$ is assumed to be a function evaluation and $p(x)$ a test on the argument value x, each taking one machine cycle. Any more complex computation g is assumed to be made of basic machine operations like $f(x)$ and $p(x)$.

So far we have only considered one form of invocation, by action in from the environment. Tasks can also invoke each other (inv_j), often as the last action of invocation, and be invoked by timers. A timer ($Timer_i$) is always ready to accept a new time period ($time_i$) after which it will timeout ($timeout_i$).

$$\begin{aligned} Timer_i \mathrel{\widehat{=}} \mu X.[X &\mathrel{\widehat{=}} time_i(t).X'(t)]\\ [X'(t) &\mathrel{\widehat{=}} \varepsilon(t).X'' + time_i(u).X'(u)]\\ [X'' &\mathrel{\widehat{=}} \overline{timeout_i}.X + time_i(u).X'(u)] \end{aligned}$$

Finally, let Lt consist of the actions inv_i for the invocation of tasks and timing actions $time_i$ and $timeout_i$. Then $Tasks =_{def} (|_{i=1}^{per+spo} Task_i \mid |_{i=1}^{per} Timer_i) \backslash Lt$ and $Resources =_{def} |_{i=1}^{pro} Processor_i$.

3.2 Scheduling

The scheduler maps tasks to resources. Define the following sets of actions: Lts for communication between tasks and the scheduler (req_i and rel_i), Ltr for actions between tasks and resources ($tick_i$) and Lsr for actions between the scheduler and resources (prt_k). Let $L =_{def} Lts \cup Ltr \cup Lsr$. Then using a scheduler which accepts requests (req_i) for processors, allocates tasks to processors (prt_k), and keeps an updated knowledge of available resources (rel_i), the mapping results in the process $(Tasks|Resources|Scheduler)\backslash L$.

For example, let $f : [1, spo + per] \rightharpoonup [1, pro] \cup \{\bot, \top\}$ record the status of tasks: if $f(i) = \bot$ then $Task_i$ is waiting for an invocation; if $f(i) = \top$ then it is active but not being executed; and if $f(i) \in [1, pro]$ then it is under execution on $Processor_{f(i)}$. If $f(i) = \top$ then we say that $Task_i$ is suspended and if $k \notin rng(f)$ then $Processor_k$ is idle. Let initially $f_0(i) = \bot$. The relative 'importance' of tasks is represented by their priorities $\pi : [1, spo + per] \rightharpoonup N$. Using priorities, $Task_i$ will be allocated a processor only if no $Task_j$ of higher priority needs one, as represented by the scheduler $\mu X(f_0).[X(f) \mathrel{\hat{=}} \cdots]$ where predicate $max(i, f) =_{def} f(i) = \top \wedge (f(j) = \top \Rightarrow \pi(i) \geq \pi(j))$ and

$$\begin{aligned} X(f) \mathrel{\hat{=}} & \textstyle\sum_{f(i)=\bot} req_i.X(f[\top/i]) + \\ & \textstyle\sum_{f(i)\notin\{\bot,\top\}} rel_i.X(f[\bot/i]) + \\ & \textstyle\sum_{max(i,f)} \sum_{k\notin rng(f)} \overline{prt}_k(i).X(f[k/i]) \end{aligned}$$

Once allocated, the priority-based scheduler will let a task run until its completion. A *pre-emptive* scheduler, in contrast, may replace the task ($Task_j$) with the lowest priority among all executing tasks, $min(j, f)$, by the task ($Task_i$) with the highest priority among suspended tasks, $max(i, f)$. Then predicate $min(j, f) =_{def} rng(f) = [1, pro] \wedge (f(k) \in [1, pro] \Rightarrow \pi(j) \leq \pi(k))$ and such a scheduler is $\mu X(f_0).[X(f) \mathrel{\hat{=}} \cdots]$ where

$$\begin{aligned} X(f) \mathrel{\hat{=}} & \textstyle\sum_{f(i)=\bot} req_i.X(f[\top/i]) + \\ & \textstyle\sum_{f(i)\notin\{\bot,\top\}} rel_i.X(f[\bot/i]) + \\ & \textstyle\sum_{max(i,f)} \sum_{k\notin rng(f)} \overline{prt}_k(i).X(f[k/i])] + \\ & \qquad \textstyle\sum_{min(j,f)} \overline{prt}_{f(j)}(i).X(f[f(j)/i, \top/j]) \end{aligned}$$

For pre-emptive scheduling of independent periodic tasks, an optimal allocation of static priorities is the so-called rate-monotonic order, inverse-proportional to the tasks' invocation periods: if $period_i \leq period_j$ then $\pi(i) \geq \pi(j)$.

3.3 Communication

Assume that tasks communicate asynchronously through shared objects. In its simplest form, such an *Object* provides some data storage that can be read using two actions (say request and completion) and modified, each with some delay *delay*. Let $\bot$ be the initial value.

$$\begin{aligned} Object =_{def} \mu X(\bot).[X(x) \mathrel{\hat{=}} & rd.\varepsilon(delay).\overline{rd}(x).X(x) + \\ & wt(y).\varepsilon(delay).X(y)] \end{aligned}$$

Suppose we have *obj* such objects and let us redefine *Resources* to take account of both kinds of resources: $Resources =_{def} |_{i=1}^{pro} Processor_i \,||_{i=1}^{obj} Object_i$. But with mutual exclusion over shared objects, a lower priority task may suspend a higher priority task. For example, if $\pi(i) > \pi(j) > \pi(k)$, $Task_k$ may secure exclusive access to the shared object before $Task_i$. Then $Task_i$ has to wait until $Task_k$ completes and $Task_j$ may be executed instead (*priority inversion*).

Assume that in order to use a shared object $Object_j$, $Task_i$ first requests access from the scheduler by the action $\overline{req}_i(j)$; it will later perform $\overline{rel}_i(j)$ to release the object. This requires some additional forms of declarations for $Task_i$:

$$\begin{aligned} X(j) &\mathrel{\widehat{=}} \overline{req}_i(j).\overline{rd}_j.rd_j(x).\overline{rel}_i(j).X'(x) \\ X(j,x) &\mathrel{\widehat{=}} \overline{req}_i(j).\overline{wt}_j(x).\overline{rel}_i(j).X' \end{aligned}$$

As $tick_i$ represents delays caused by the underlying processors, it need not appear in these declarations: delays there are only caused by the sharing of objects (resolved by the scheduler) and the time it takes to access them.

The Immediate Ceiling Priority Inheritance Protocol solves the problem by assigning a priority to an object that is the maximum of the priorities of all tasks that share the object $\rho : [1, obj] \rightarrow N$. Then each time a task obtains access to an object, its priority is immediately raised to the ceiling level. For a given object, let the function $g : [1, obj] \rightarrow N \cup \{\bot\}$ return either the original priority of the task accessing the object or $\bot$ if there is no such task. Initially, $g_0(l) = \bot$. The protocol can now be added to the scheduler, as below.

$$\begin{aligned} X(f,g,\pi) \mathrel{\widehat{=}} & \textstyle\sum_{f(i)=\bot} req_i.X(f[\top/i],g,\pi) + \\ & \textstyle\sum_{f(i)\notin\{\bot,\top\}} rel_i.X(f[\bot/i],g,\pi) + \\ & \qquad \textstyle\sum_{g(j)=\bot} req_i(j).X(f,g[\pi(i)/i],\pi[\rho(i)/i]) + \\ & \qquad \textstyle\sum_{g(j)\neq\bot} rel_i(j).X(f,g[\bot/i],\pi[g(i)/i]) + \\ & \textstyle\sum_{max(i,f)} \sum_{k\notin rng(f)} \overline{prt}_k(i).X(f[k/i],g,\pi)] + \\ & \qquad \textstyle\sum_{min(j,f)} \overline{prt}_{f(j)}(i).X(f[f(j)/i,\top/j],g,\pi) \end{aligned}$$

3.4 Distribution

The mapping between tasks and resources has so far assumed use of a centralized scheduler. Suppose instead that tasks are partitioned between $nd > 0$ nodes ($Node^i$) arranged into a logical ring and connected by a multiple-access network (*Network*). Each node provides computing resources like clocks, memory and processors and each has a local scheduler. The actions at $Node^i$ will be distinguished by the superscript i, $Node^i =_{def} (Tasks^i|Resources^i|Scheduler^i)\backslash L^i$.

Suppose that each task has a local object ($Object_i$ for $Task_i$) to hold the sequences of messages to be sent. The sending of a message m is then represented by the following declarations:

$$\begin{aligned} X(m) &\mathrel{\widehat{=}} \overline{req}_i(i).\overline{rd}_i.rd_i(s).\overline{rel}_i(i).X_1(m,s) \\ X_1(m,s) &\mathrel{\widehat{=}} tick_i.X_2(s:m) \\ X_2(s) &\mathrel{\widehat{=}} \overline{req}_i(i).\overline{wt}_i(s).\overline{rel}_i(i).X_2' \end{aligned}$$

Since we assume that tasks cannot voluntarily suspend themselves, a task can only invoke a remote task and write to a remote object. Therefore data messages have the form $n.c.j.v$ where $n \in [1, nd]$ is a node ($n \neq i$), c is either $invoke$ or $write$, j identifies the task ($c = invoke$, $j \in [1, spo^n + per^n]$) or object ($c = write$, $j \in [1, obj^n]$), and v is the value passed to the task or written to the object.

Unlike the scheduling of local resources (processors or objects) between tasks, the scheduling of network traffic (deciding which node is allowed to transmit and for how long) cannot be done centrally. We shall use a simple protocol based on a circulating token ($token$). After receiving the token, $Node^i$ may transmit one message (the first message of the highest priority task) before passing the token to $Node^{i+1}$. A task $Task_1^i$ with the highest priority is used to implement the protocol on each node. This task is sporadic, invoked by action in_1^i and producing a result by $\overline{out}_1^i$. Given a function $h(i)$ on $[1, nd]$ which returns either $\perp$ or the last message received from $Node_i$ ($h_0(i) = \perp$), the network is defined below.

$$Network =_{def} \mu X(h_0).[X(h) \mathrel{\widehat{=}} \textstyle\sum_{h(j)=\perp} out_1^j(x).X(h[x/j]) + \sum_{h(j)\neq\perp} \overline{in}_1^{(j+1) \bmod nd}(h(j)).X(h[\perp/j])]$$

Let tasks be ordered according to decreasing value of priority, with the tasks of the same priority ordered by the number (j). Given $j \in [1, spo^n + per^n]$, let $suc(j)$ return an immediate successor of j or $\perp$ if there is no successor. Then π determines not only the importance of tasks but also of messages: $Task_1^i =_{def} \mu X.[X \mathrel{\widehat{=}} \cdots]$ and $Task_1^i(token) =_{def} \mu Y.[X \mathrel{\widehat{=}} \cdots]$ with the declarations below.

X	$\mathrel{\widehat{=}} in_1^i(x).\overline{req}_1^i.X_1(x)$	invocation
$X_1(x)$	$\mathrel{\widehat{=}} tick_1^i.if\ x \neq token\ then\ X_2(x)\ else\ Y_1(0)$	token received?
$X_2(x)$	$\mathrel{\widehat{=}} tick_1^i.if\ x \neq i.c.j.v\ then\ X_3(x)\ else\ X_4(c,j,v)$	for us?
$X_3(x)$	$\mathrel{\widehat{=}} \overline{rel}_1^i.\overline{out}_1^i(x).X$	forward x
$X_4(c,j,v)$	$\mathrel{\widehat{=}} tick_1^i.if\ c = invoke\ then\ X_5(j,v)\ else\ X_6(j,v)$	invocation?
$X_5(j,v)$	$\mathrel{\widehat{=}} \overline{rel}_1^i.\overline{inv}_j(v).X$	invoke j
$X_6(j,v)$	$\mathrel{\widehat{=}} \overline{req}_1^i(j).\overline{wr}_j^i(v).\overline{rel}_1^i(j).X_7$	write to j
X_7	$\mathrel{\widehat{=}} \overline{rel}_1^i.X$	release
$Y_1(j)$	$\mathrel{\widehat{=}} tick_1^i.Y_2(suc(j))$	smaller priority
$Y_2(j)$	$\mathrel{\widehat{=}} tick_1^i.if\ j = \perp\ then\ Y_3\ else\ Y_4(j)$	the smallest?
Y_3	$\mathrel{\widehat{=}} \overline{rel}_1^i.\overline{out}_1^i(token).X$	release and forward
$Y_4(j)$	$\mathrel{\widehat{=}} \overline{req}_1^i(j).\overline{rd}_j^i.rd_j^i(s).\overline{rel}_1^i(j).Y_5(j,s)$	read j
$Y_5(j,s)$	$\mathrel{\widehat{=}} tick_1^i.if\ s = \varepsilon\ then\ Y_1(j)\ else\ Y_6(j,s)$	no messages?
$Y_6(j,s)$	$\mathrel{\widehat{=}} tick_1^i.Y_7(j, s_0, s)$	take the head
$Y_7(j,m,s)$	$\mathrel{\widehat{=}} tick_1^i.Y_8(j,m,s')$	take the tail
$Y_8(j,m,s)$	$\mathrel{\widehat{=}} \overline{req}_1^i(j).\overline{wr}_j^i(s).\overline{rel}_1^i(j).Y_9(m)$	write the tail
$Y_9(m)$	$\mathrel{\widehat{=}} \overline{rel}_1^i.\overline{out}_1^i(m).\overline{out}_1^i(token).X$	release and send

Let $Node^i(token)$ be like $Node^i$ but with $Task_1^i(token)$ replacing $Task_1^i$ and let $Node^1$ hold the token initially. Then for Lc containing actions in_1^i and out_1^i, we have the distributed system $(Node^1(token)|Node^2|\cdots|Node^{nd}|Network)\backslash Lc$.

3.5 Specification, Verification and Equation-Solving

So far we have shown how a simple timed process algebra framework can be used to build a fairly general model for communicating systems which is capable of representing resource-limited executions. We shall now show how the timed and untimed properties of such systems can be specified and verified.

Consider two actions, a and b, for which it is required that whenever a occurs, b occurs at most d later ($T_1(a,b,d)$) or d earlier ($T_2(a,b,d)$).

$$\begin{array}{lll} T_1(a,b,d) =_{def} \nu Z.[Z & \widehat{=}[a]Z'(0) \wedge \bigwedge_{\alpha \neq a}[\alpha]Z] \\ & [Z'(t) \widehat{=} \langle b \rangle tt \wedge [b]Z \vee \bigvee_{\alpha}\langle \alpha \rangle tt \wedge \\ & \quad \bigwedge_{c \neq b}[c]Z'(t) \wedge \bigwedge_{u \leq d-t}[\varepsilon(u)]Z'(t+u)] \\ T_2(a,b,d) =_{def} \nu Z.[Z & \widehat{=}[a]Z'(0) \wedge \bigwedge_{\alpha \neq a}[\alpha]Z] \\ & [Z'(t) \widehat{=} [b]ff \wedge \bigwedge_{c \neq b}[c]Z'(t) \wedge \bigwedge_{u \geq 0}[\varepsilon(u)]Z'(t+u) \mid t \leq d] \\ & [Z'(t) \widehat{=} [b]Z \wedge \bigwedge_{\alpha \neq b}[\alpha]Z'(t) \mid t > d] \end{array}$$

A simple functional property, in contrast, would state that if the value x received by action $a(x)$ satisfies a pre-condition $pre(x)$, the value y of $b(y)$ must satisfy a post-condition $post(x,y)$. This, plus the timing requirement that $b(y)$ occurs no later than d after $a(x)$, is defined by the predicate below.

$$\begin{array}{l} T_1'(a(x) : pre(x), b(y) : post(x,y), d) =_{def} \\ \quad \nu Z.[Z \quad \widehat{=} \bigwedge_{x:pre(x)}[a(x)]Z'(0,x) \wedge \bigwedge_{x:\neg pre(x)}[a(x)]Z] \wedge \bigwedge_{\alpha \neq a}[\alpha]Z] \\ \qquad [Z'(t,x) \widehat{=} \bigvee_{y:post(x,y)}(\langle b(y) \rangle tt \wedge [b(y)]Z) \vee \bigvee_{\alpha}\langle \alpha \rangle tt \wedge \\ \qquad\quad \bigwedge_{c \neq b}[c]Z'(t) \wedge \bigwedge_{u \leq d-t}[\varepsilon(u)]Z'(t+u)] \end{array}$$

It is also easy to define that a occurs with period p and *jitter* d, relative the beginning of each period. Predicates such as this can be used as the building blocks for typical transactions, relating the timing and values of task interactions. A transactions will typically relate the input to a task ($Task_i$) with the output from another task ($Task_j$) which may not be on the same node. Let $Task_i$ be located at $Node^n$, $Task_j$ at $Node^m$ and after the action $in_i^n(x)$ in which x satisfies the pre-condition $pre(x)$, action $\overline{out}_j^m(y)$ must occur no earlier than d_1 and no later than d_2 and with y satisfying the post-condition $post(x,y)$.

$$\begin{array}{l} Transaction =_{def} T_1'(in_i^n(x) : pre(x), \overline{out}_j^m(y) : post(x,y), d_1) \wedge \\ \qquad \bigwedge_{x,y} T_2(in_i^n(x), \overline{out}_j^m(y), d_2) \end{array}$$

With limited computing resources and in the absence of assumptions about how often in_i^n arrives, it is in general impossible to meet this transaction. Let d_2 be the minimum inter-arrival time for action in_i^n:

$$Assumption =_{def} \bigwedge_{x,y} T_2(in_i^n(x), in_i^n(y), d_2)$$

Then given a real-time system ($System$), the properties of transactions must only be verified when the assumptions are satisfied.

$$System \models Assumptions \Rightarrow Transactions$$

And if *System* has the form described earlier, verification will take a full account of the constraints imposed by the underlying hardware, for centralized and distributed systems respectively.

$$(Tasks|Resources|Scheduler)\backslash L \models Assumptions \Rightarrow Transactions$$
$$(Node^1(token)||_{i=2}^{nd} Node^i|Network)\backslash Lc \models Assumptions \Rightarrow Transactions$$

One more advantage of representing resource constraints syntactically is the possibility of finding a feasible scheduler (if one exists) automatically, as the well-known equation-solving problem. The problem has attracted some attention [21, 17] and algorithmic solutions have been proposed and implemented [10].

4 Fault-Tolerance for Unlimited Resources

In Section 2 we introduced a general framework for describing and reasoning about distributed and real-time systems and in Section 3 showed how to represent and verify systems which can only rely on limited (in terms of the number and speed) set of hardware resources. And we made it very specific of how hardware (processors, memory, clocks or communication media) should behave in order for properties of the overall system to hold. We now show how to reason about systems that are designed to sustain anticipated hardware failures, to show that they are provably fault-tolerant. We continue describing faults and their effect on the semantics of TCCS, and then show how we can prove fault-tolerance, for given assumption about faults and first for unlimited resources.

4.1 Faults and their Effect

The fault-tolerance of a system is often verified by syntactically transforming it into its *fault-affected* version and then verifying its properties as if no faults are present [18]. This method allows standard techniques to be used for proving fault-tolerance, so we begin by examining how it can be used in our logic.

For a process Q, assume that a 'faulty' declaration Ψ, in general different from 'normal' declarations being part of the syntax of Q, is used to specify anticipated faults. Let Q be transformed into $\mathcal{T}(Q,\Psi)$ to represent the effects of such faults. The transformation is defined as follows.

$$\begin{array}{ll}
\mathcal{T}(0,\Psi) & =_{def} 0 \\
\mathcal{T}(\mu X.\Delta,\Psi) & =_{def} \mu X.(\Delta \oplus \Psi) \\
\mathcal{T}(\alpha.Q,\Psi) & =_{def} \alpha.\mathcal{T}(Q,\Psi) \\
\mathcal{T}(Q_1+Q_2,\Psi) & =_{def} \mathcal{T}(Q_1,\Psi)+\mathcal{T}(Q_2,\Psi) \\
\mathcal{T}(Q_1|Q_2,\Psi) & =_{def} \mathcal{T}(Q_1,\Psi)|\mathcal{T}(Q_2,\Psi) \\
\mathcal{T}(Q \setminus L,\Psi) & =_{def} \mathcal{T}(Q,\Psi) \setminus L \\
\mathcal{T}(Q[g],\Psi) & =_{def} \mathcal{T}(Q,\Psi)[g]
\end{array}$$

Assume that $\mathcal{T}(Q,\Psi)$ is well-defined, i.e. all constants declared, and since faults are autonomous, all expressions $[\![\Psi]\!](X)$ are either prefixed by τ or are

a summation of such expressions. Such a 'faulty' declaration Ψ is generated by the abstract syntax $\Psi ::= \tau \odot \Delta \mid \Psi \oplus \Psi$. Some examples are a processor which may decide to tick early, a timer which may timeout late, a shared object which sometimes fails to remember a written value and a network which may lose messages. Such faults are represented by the declarations below.

$$\begin{array}{lll}
\Psi_{processor} & =_{def} & \tau \odot [X(i) \ \widehat{=} \ prt_k(j).X(j) + \varepsilon(speed_k - 1).\overline{tick}_i.X(i)] \\
\Psi_{timer} & =_{def} & \tau \odot [X'' \ \widehat{=} \ \varepsilon(1).\overline{timeout}_i.X + time_i(u).X'(u)] \\
\Psi_{object} & =_{def} & \tau \odot [X(x) \widehat{=} rd.\varepsilon(delay).\overline{rd}(x).X(x) + wt(y).\varepsilon(delay).X(x)] \\
\Psi_{network} & =_{def} & \tau \odot [X(h) \widehat{=} X(h[\bot/j])]
\end{array}$$

However, given a process Q, a specification Ψ of faults and a property F that must hold despite these faults, verifying $\mathcal{T}(Q,\Psi) \models F$ is not sufficient to prove that Q is fault-tolerant [12]. It is necessary to take into account that faults are unpredictable: after proving correctness for a number of anticipated faults, correctness for any subset of these fault must be (provably) guaranteed. This, however, is not the case for Q and Ψ below because $\mathcal{T}(Q,\Psi) \models F$ (in the presence of all faults) but $Q \not\models F$ (in the absence of faults).

$$\begin{array}{lll}
Q =_{def} \mu X.\, [X \widehat{=} b.X''][X' \widehat{=} a.X'' + b.X'' + \tau.X'''] & \Psi =_{def} \tau \odot [X \widehat{=} X'] \\
\qquad\qquad [X'' \widehat{=} b.X][X''' \widehat{=} a.X'' + \tau.X] & F =_{def} [\varepsilon]\langle a\rangle tt
\end{array}$$

The reason is action a which is only possible in the presence of faults. But even if $Q \models F$ and $\mathcal{T}(Q, \Psi \oplus \Phi) \models F$, F may no longer hold if only some of the faults are present ($\mathcal{T}(Q,\Psi) \not\models F$), as below.

$$\begin{array}{lll}
Q =_{def} \mu X.\, [X \ \widehat{=} \ \tau.X' + b.X''] & \Psi =_{def} \tau \odot [X' \widehat{=} X'''] \\
\qquad\qquad [X' \ \widehat{=} \ a.X'' + b.X''] & \Phi =_{def} \tau \odot [X''' \widehat{=} X] \\
\qquad\qquad [X'' \widehat{=} b.X][X''' \widehat{=} a.X''] & F =_{def} [\varepsilon]\langle b\rangle tt
\end{array}$$

This is because the faults Ψ may result in the state X''' but then action b is only possible in the presence of Φ. The property of *fault-monotonicity* is not assured in this logic or in many other semantical theories for branching time (bisimulations, testing equivalence, etc).

To define a fault-monotonic version of the logic, however, we need to first define the fault-affected semantics of the language explicitly. We do so using relation $\overset{\gamma}{\underset{\Psi}{\longmapsto}}$ for Ψ-affected transitions, with $\overset{\gamma}{\underset{\Psi}{\longmapsto}}$ defined similarly to $\overset{\gamma}{\longrightarrow}$, but with one additional transition rule:

$$\frac{E_i \overset{\gamma}{\underset{\Psi}{\longmapsto}} E_i', i \in I}{E \overset{\gamma}{\underset{\Psi}{\longmapsto}} E'} \text{ for all } \frac{E_i \overset{\gamma}{\longrightarrow} E_i', i \in I}{E \overset{\gamma}{\longrightarrow} E'} \text{ and } \gamma \in \{\alpha, \eta\} \text{ in Table 2}$$

$$\frac{[\![\Psi]\!](X)\{\widetilde{\mu Y.\Delta}/\widetilde{Y}\} \overset{\tau}{\underset{\Psi}{\longmapsto}} P'}{\mu X.\Delta \overset{\tau}{\underset{\Psi}{\longmapsto}} P'} \ (X \in dom(\Psi))$$

4.2 Proving Fault-Tolerance

Any transition which is possible in the absence of faults ($\longrightarrow$) is also possible in their presence ($\stackrel{}{\underset{\Psi}{\longmapsto}}$). But in a fault-monotonic version of the logic, transitions which are *only* possible in the presence of faults require special attention as they must be tolerated when they occur but, like faults, they cannot be relied upon to occur. The first step towards this is to remove negation from the logic.

$$\begin{array}{l} Fe ::= tt \mid ff \mid Z \mid \bigvee_{i\in I} Fe \mid \bigwedge_{i\in I} Fe \mid \langle\widehat{\alpha}\rangle Fe \mid [\widehat{\alpha}]Fe \\ \nabla \ ::= [Z \mathrel{\widehat{=}} Fe] \mid [Z \mathrel{\widehat{=}} Fe]\nabla \\ F \ ::= tt \mid ff \mid \nu Z.\nabla \mid \mu Z.\nabla \mid \bigvee_{i\in I} F \mid \bigwedge_{i\in I} F \mid \langle\widehat{\alpha}\rangle F \mid [\widehat{\alpha}]F \end{array}$$

The next step is to remove the symmetry between modalities, so that $\langle\alpha\rangle F$ is verified according to the transitions $\longrightarrow$ and $[\alpha]F$ according to $\stackrel{}{\underset{\Psi}{\longmapsto}}$; the latter will ensure that such transitions are tolerated and the former that they are not relied upon. Given Ψ, the semantics is below ($Q \models_\Psi F$ iff $Q \in [\![F]\!]$).

$$\begin{array}{lll} [\![tt]\!]_\delta =_{def} \mathcal{P} & [\![ff]\!]_\delta =_{def} \emptyset & [\![Z]\!]_\delta =_{def} \delta(Z) \\ [\![\bigwedge_{i\in I} F_i]\!]_\delta =_{def} \bigcap_{i\in I} [\![F_i]\!]_\delta & & [\![\mu Z.\nabla]\!]_\delta =_{def} \bigcap\{\delta' \mid [\![\nabla]\!]_{\delta'} \subseteq \delta'\}(Z) \\ [\![\bigvee_{i\in I} F_i]\!]_\delta =_{def} \bigcup_{i\in I} [\![F_i]\!]_\delta & & [\![\nu Z.\nabla]\!]_\delta =_{def} \bigcup\{\delta' \mid \delta' \subseteq [\![\nabla]\!]_{\delta'}\}(Z) \\ [\![\langle\widehat{\alpha}\rangle F]\!]_\delta =_{def} \{P \mid \exists_{P',t} P \xrightarrow{t} P' \wedge \widehat{t} = \widehat{\alpha} \wedge P' \in [\![F]\!]_\delta\} & & \\ [\![[\widehat{\alpha}]F]\!]_\delta =_{def} \{P \mid \forall_{P',t} (P \stackrel{t}{\underset{\Psi}{\longmapsto}} P' \wedge \widehat{t} = \widehat{\alpha}) \Rightarrow P' \in [\![F]\!]_\delta\} & & \end{array} \quad (3)$$

This treatment of modalities corresponds to the way the refinement preorder of Modal Process Logic [16] received its modal characterisation [14]. The motivation there is different: $\longrightarrow$ are transitions of the specification that the implementation must perform, and $\stackrel{}{\underset{\Psi}{\longmapsto}}$ are transitions that may or may not be performed. (MPL and fault-tolerance are discussed again in Section 6).

5 Fault-Tolerance for Limited Resources

A realistic analysis of the timing properties of a system must take into account the limitations of the underlying hardware. This is even more needed if hardware failures are to be tolerated. Fault-tolerance requires redundancy – additional components (hardware redundancy), instructions (software redundancy) or executions (time redundancy) – and redundancy requires resources and time. Resources must be assigned when a fault occurs (e.g. for rollback recovery) and also to enable run-time recovery, e.g. for periodic checkpointing and for voting on the outcome of N-modular executions.

We shall now combine consideration of resource limitations and faults and show how the timing properties of fault-tolerant and resource-limited systems can be analysed. A major issue, like before, is the allocation of tasks to resources. But now we shall use dynamic allocation according to the urgency of tasks and availability of resources.

5.1 Proving Fault-Tolerance for Bounded Resources

As before, let a system consist of a number of tasks, $Tasks$, some of them periodic and others sporadic, each with its own timer, executed on a centralized set of resources, $Resources$, including processors and protected shared objects. Let a scheduler, $Scheduler$, map tasks into resources in a way that ensures that the timing constraints $Timing$ are met despite hardware failures $\Psi_{resources}$.

$$(Tasks|Resources|Scheduler)\backslash L \models_{\Psi_{resources}} Timing$$

$Timing$ may contain a number of requirements but with limited computing resources and with no assumptions about how often sporadic tasks arrive, to satisfy them may not be possible. But $Timing$ contains no negation (to ensure fault-monotonicity) and thus cannot express implication. This basic problem results from the nature of verifying the timing properties of resource-bound systems in the presence of faults. We shall assume that resources, $Resources$, are not shared with tasks which are part of the environment. Therefore the inter-arrival time of the sporadic tasks (perhaps invoked by these environment tasks) will never depend on failures of these resources. The solution is then to first verify assumptions in the absence of faults ($\models$) and if they hold then to also verify transactions in the presence of anticipated faults ($\models_{\Psi_{resources}}$).

$$\begin{array}{l}(Tasks|Resources|Scheduler)\backslash L \models Assumptions \quad \text{then} \\ (Tasks|Resources|Scheduler)\backslash L \models_{\Psi_{resources}} Transactions\end{array}$$

5.2 Dynamic Best-Effort Scheduling

In order to make decisions after the occurrence of a fault, a scheduler must have information about the resources available at that time. For example, consider a fail-stop assumption [25] and the actions $crash_k$ and $repaired_k$ by which a scheduler is informed of the status of $Processor_k$, assuming that repair takes time $repair_k$:

$$\begin{array}{rl}\Psi^k_{fail-stop} =_{def} \tau\odot ([X(i) & \widehat{=} \overline{fail}_k.\varepsilon(repair_k).Y] \\ [Y & \widehat{=} \overline{repaired}_k.\sum_j prt_k(j).X(j)])\end{array}$$

Let the function $g : [1, pro] \rightarrow \{\bot, \top\}$ for $k \in [1, pro]$ return $\top$ if $Processor_k$ is operative and $\bot$ otherwise (initially $g_0(k) = \top$). Then in order to non-preemptively schedule independent tasks in the presence of faults, we have the scheduler $\mu X(f_0, g_0).[X(f, g) \widehat{=} \cdots]$ where

$$\begin{array}{rl}X(f,g) \widehat{=} & \sum_{f(i)=\bot} req_i.X(f[\top/i], g)+ \\ & \sum_{f(i)\notin\{\bot,\top\}} rel_i.X(f[\bot/i], g)+ \\ & \sum_{max(i,f)\wedge k\notin rng(f)\wedge g(k)=\top} \overline{prt}_k(i).X(f[k/i], g)+ \\ & \sum_{g(k)=\top\wedge k\notin rng(f)} fail_k.X(f, g[\bot/k])+ \\ & \sum_{g(k)=\top\wedge k\in rng(f)} fail_k.X(f[\bot/f^{-1}(k)], g[\bot/k])+ \\ & \sum_{g(k)=\bot} repair_k.X(f, g[\top/k])\end{array}$$

Another consequence of the presence of faults is that the static allocation of priorities to tasks is then usually ineffective. Consider the well-known earliest-deadline-first (EDF) policy: the closer the task's deadline, the higher its priority. This policy is easy to implement for tasks with individual deadlines. Let $d : [1, spo{+}per] \rightarrow R_+$ denote such deadlines and for all i such that $f(i) \neq \bot$ (i.e. for all invoked tasks) let $h(i)$ return the time that $Task_i$ has been invoked; initially $h_0(i) = 0$. We introduce a new prefix operator $a@t.Pe$ to represent the delay before the action a is offered and assuming that Pe contains the time variable t, we have the rules $a@t.Pe \xrightarrow{a} Pe[0/t]$ and $a@t.Pe \xrightarrow{\varepsilon(d)} a@t.Pe[t + d/t]$ [26]. Finally, let the predicate $min(i, f, h, t)$ hold if among the suspended tasks, $Task_i$ is the closest to violating its deadline: $min(i, f, h, t) =_{def} f(i) = \top \wedge (f(j) = \top \Rightarrow d(i) + h(i) - t \leq d(j) + h(j) - t)$. Then EDF can be implemented by $\mu X(f_0, g_0, h_0, 0).[X(f, g, h, t) \mathrel{\widehat{=}} \cdots]$ where

$$\begin{aligned} X(f,g,h,t) \mathrel{\widehat{=}} & \textstyle\sum_{f(i)=\bot} req_i@t.X(f[\top/i], g, h[t/i], t) + \\ & \textstyle\sum_{f(i)\notin\{\bot,\top\}} rel_i@t.X(f[\bot/i], g, h, t) + \\ & \textstyle\sum_{min(i,f,h,t)\wedge k\notin rng(f)\wedge g(k)=\top} \overline{prt}_k(i)@t.X(f[k/i], g, h, t) + \\ & \textstyle\sum_{g(k)=\top\wedge k\notin rng(f)} fail_k@t.X(f, g[\bot/k], h, t) + \\ & \textstyle\sum_{g(k)=\top\wedge k\in rng(f)} fail_k@t.X(f[\bot/f^{-1}(k)], g[\bot/k], h, t) + \\ & \textstyle\sum_{g(k)=\bot} repair_k@t.X(f, g[\top/k], h, t) \end{aligned}$$

The EDF policy is optimal for independent tasks on a single fault-free processor and a *best effort* policy in general [23].

5.3 Dynamic Planning-Based Scheduling

A *planning-based* scheduler, in contrast, will only schedule a task if its deadlines can be guaranteed. Let each task request a processor by sending an upper bound $bound(x)$ on the number of basic machine cycles to complete an invocation (x is a parameter) and let acceptance and rejection of tasks be represented by the actions acc_i and rej_i respectively. Then for sporadic tasks we have:

$$\begin{aligned} Task_i =_{def} \mu X.\ & (tick_i \odot \Delta) \\ & [X \mathrel{\widehat{=}} in_i(x).\overline{req_i}(bound(x)).(acc_i.Y(x) + rej_i.X)] \\ & [Z(z) \mathrel{\widehat{=}} \overline{rel_i}.\overline{out_i}(z).X] \end{aligned}$$

A planning-based scheduler will maintain a schedule of all tasks that will guarantee their timely completion provided no processors fail in the meanwhile. The schedule is represented by the function $h : [1, pro] \rightarrow [1, per + spo]^*$ which returns the sequence of tasks that are scheduled to be executed on each processor ($h(k)_0$ is currently executed on $Processor_k$ and initially $h_0(k) = \varepsilon$). In addition, we apply $b : [1, per + spo] \rightarrow N$ to return the upper bound on the number of machine cycles for the current invocation of each active task (initially $b_0(i) = 0$).

Each time a task completes, the next task is taken for execution and when a new task arrives, the scheduler will try to accommodate its execution in the existing schedule. This is done by looking for an operative $Processor_k$ which is

fast enough to guarantee the additional task's deadline (c is the number of cycles and sum returns the sum of all numbers in the sequence): $fst(i,k,h,c) =_{def} (sum(h(k))+c)*speed_k \leq d(i)$. The task is accepted *iff* such a processor exists. In case a processor fails, the scheduler will try to relocate all its tasks for execution on other operative processors. This, however, may not always succeed and the scheduler then enters a degraded mode of operation in which tasks which cannot be accommodated will be dropped from the execution. Each time this happens, the action $\overline{degrade}$ is performed, announcing the number of the task.

$$
\begin{array}{lll}
X(f,g,h,b) \mathrel{\widehat{=}} & \sum_{f(i)=\bot} & req_i(c).X_1(f,g,h,b,c,i)+ \\
& \sum_{f(i)\notin\{\bot,\top\}} & rel_i.X_2(f[\bot/i],g,h[h(f(i))'/f(i)],b,f(i))+ \\
& \sum_{g(k)=\top} & fail_k.X_3(f,g[\bot/k],h,b,k)+ \\
& \sum_{g(k)=\bot} & repair_k.X(f,g[\top/k],h,b) \\
X_1(f,g,h,b,c,i) \mathrel{\widehat{=}} & \sum_{g(k)=\top \Rightarrow \neg fst(i,k,h,c)} & \overline{rej}_i.X(f,g,h,b)+ \\
& \sum_{g(k)=\top \wedge fst(i,k,h,c)} & \overline{acc}_i.X_2(f[\top/i],g,h[h(k):i/k],b[c/i],k) \\
X_2(f,g,h,b,k) \mathrel{\widehat{=}} & \multicolumn{2}{l}{if\ h(k)=\varepsilon \vee f(h(k)_0) \neq \top} \\
& \multicolumn{2}{l}{then\ X(f,g,h,b)\ else\ \overline{prt}_k(h(k)_0).X(f[k/h(k)_0],g,h,b)} \\
X_3(f,g,h,b,k) \mathrel{\widehat{=}} & \multicolumn{2}{l}{if\ h(k)=\varepsilon\ then\ X(f,g,h,b)\ else\ X_4(f,g,h,b,k,h(k)_0)} \\
X_4(f,g,h,b,k,i) \mathrel{\widehat{=}} & \sum_{g(l)=\top \wedge fst(i,l,h,b(i))} & X_2'(f,g,h[h(k)'/k,h(l):i/l],b,l)+ \\
& \sum_{g(l)=\top \Rightarrow \neg fst(i,l,h,b(i))} & \overline{degrade}(i).X3(f,g,h[h(k)'/k],b,k) \\
X_2'(f,g,h,b,k) \mathrel{\widehat{=}} & \multicolumn{2}{l}{if\ h(k)=\varepsilon \vee f(h(k)_0) \neq \top} \\
& \multicolumn{2}{l}{then\ X_3(f,g,h,b)\ else\ \overline{prt}_k(h(k)_0).X_3(f[k/h(k)_0],g,h,b)}
\end{array}
$$

As we can see, the planning-based policy above will only provide guarantees if no failure occurs after tasks are allocated but will otherwise degrade gracefully if some tasks cannot be accommodated. Under sufficiently strong assumptions it may be possible to provide guarantees in the presence of any faults, but the issues of feasibility (assumptions) and utilization (resources) may make such a solution impractical. The graceful degradation, however, will make it possible to share the load among the different nodes of a distributed system, and to relocate the tasks for which the deadlines cannot be guaranteed. We have already shown how to schedule network traffic to consider the urgency of messages. A similar replication of objects and tasks can also be used to ensure resiliency to node and memory failures. The issue is then to ensure that the replicas are consistent.

6 Conclusions

To analyse the timing properties of a distributed system, it is essential to consider the limitations of the resources of the system and the way resources are allocated to tasks. The existing formal techniques are either based on the maximal parallelism assumption or provide very basic means of resolving competition for resources, by statically assigning priorities to actions. If, in addition, hardware failures are to be considered, then to statically determine the task execution order is usually inappropriate. In this paper, we have shown how the simple

framework of Timed CCS can be used for a general model for resource-based executions. We have also demonstrated the use of different techniques for task scheduling – non-pre-emptive and pre-emptive, static and dynamic, best-effort and planning-based – and showed how to handle priority inversion and to schedule network traffic.

Since faults are unpredictable, reasoning about fault-tolerance must be fault-monotonic: after proving correctness for a number of faults, correctness for some of them must be guaranteed. Most techniques for provable fault-tolerance are based on a syntactic representation of faults. Using modal μ-calculi and the additional transitions to model the effects of faults, we have demonstrated that this common technique will not ensure fault-monotonicity. The first step in a solution is to clearly separate design decisions and environment assumptions and this was done by providing the explicit fault-affected semantics of the process language. The semantics is used in the second step, where the logic is refined into its fault-monotonic version using the timed and modal extensions of the Hennessy-Milner logic. The logic can verify fault-tolerance and we have demonstrated that it can be used to specify simple transactions.

Our work has been based on the timed extension of CCS, Timed CCS [26], and this was chosen as the simplest framework to suit our purposes. TCCS has been further extended to allow loose specifications, in Timed Modal Specifications [7] which follow Modal Process Logic [16]. It is possible to use MPL to specify and verify fault-tolerance [4]. MPL and its refinement ordering would also permit fewer faults than the maximum to occur, applying admissible transitions to specify them. But without separating design constraints (transitions which are admissible but unnecessary) and environment assumptions (transitions which model faults), MPL cannot, without risking realizability problems [1, 2], support refinement towards an increasing number of faults. As new design decisions are made and the need for new hardware or the higher reliability arise, it may be necessary to tolerate new faults that could not have been anticipated earlier. For untimed systems and unlimited resources, this was described in [12]; for timed systems and limited resources, this will be subject of a companion paper. The idea is to provide two ways of refinement to take account of an increasing number of anticipated faults: the rich-man's refinement proceeds to tolerate all anticipated faults, 'creating' new resources whenever needed to satisfy deadlines; the poor-man's refinement proceeds until the level of redundancy required exceeds what is available in the set of resources.

References

1. M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.
2. M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable specifications of reactive systems. *LNCS*, 372:1–17, 1989.
3. A.A. Bertossi and L.V. Mancini. Scheduling algorithms for fault-tolerance in hard-real-time systems. *Real-Time Systems*, 7(3):229–245, 1994.

4. A. Borjesson, K.G. Larsen, and A. Skou. Generality in design and compositional verification using TAV. *Formal Methods in System Design*, 6(3):239–258, 1995.
5. A. Burns and A. Wellings. A computational model for fixed priority scheduling. In M. Joseph, editor, *Real-Time Systems: Specification, Verification and Analysis*. Prentice-Hall, 1995.
6. J. Camilleri and G. Winskel. CCS with priority choice. *Information and Computation*. 116:26–37, 1995.
7. K. Cerans, J.C. Godskesen, and K.G. Larsen. Timed modal specifications. *LNCS*, 715:253–267, 1993.
8. R. Cleaveland and M. Hennessy. Priorities in process algebras. *Information and Computation*, 87:58–77, 1990.
9. R. Gerber and I. Lee. A resource-based prioritized bisimulation for real-time systems. *Information and Computation*, 113:102–142, 1994.
10. J.C. Godskesen et al. *Epsilon - User's Manual*. Department of Mathematics and Computer Science. University of Aalborg, 1993.
11. M. Hennessy and T. Regan. A process algebra for timed systems. Technical report, University of Sussex, 1991.
12. T. Janowski. *Bisimulation and Fault-Tolerance*. PhD thesis, Department of Computer Science, University of Warwick, 1995.
13. K. Larsen. Proof systems for Hennessy-Milner logic with recursion. *LNCS*, 299:215–230, 1988.
14. K. Larsen. Modal specifications. *LNCS*, 407, 1990.
15. K.G. Larsen. *Context-Dependent Bisimulation Between Processes*. PhD thesis, University of Edinburgh, Scotland, 1986.
16. K.G. Larsen and B. Thomsen. A modal process logic. In *Proc. 3rd Annual Symposium on Logic in Computer Science*, pages 203–210, 88.
17. X. Liu. *Specification and Decomposition in Concurrency*. PhD thesis, Department of Mathematics and Computer Science. University of Aalborg, 1992.
18. Z. Liu and M. Joseph. Transformations of programs for fault-tolerance. *Formal Aspects of Computing*, 4:442–469, 1992.
19. R. Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.
20. F. Moller and C. Tofts. A temporal calculus of comm. systems. *LNCS*, 458, 90.
21. J. Parrow. Submodule construction as equation solving in CCS. *Theoretical Computer Science*, 68:175–202, 89.
22. P. Pleinevaux. Real-time fault tolerant operation of the 802.5 token ring. *Real-Time Systems*, 8:79–91, 1995.
23. K. Ramamritham. Dynamic priority scheduling. In M. Joseph, editor, *Real-Time Systems: Specification, Verification and Analysis*. Prentice-Hall, 1995.
24. A. Salwicki and T. Müldner. On the algorithmic properties of concurrent programs. *LNCS*, 125, 1981.
25. R.D. Schlichting and F.B. Schneider. Fail stop processors: An approach to designing fault-tolerant computing systems. *ACM Trans. on Comp. Sys.*, 1(3), 1983.
26. Wang Yi. *A Calculus of Real Time Systems*. PhD thesis, Department of Computer Science. Chalmers University of Technology, 1991.

This article was processed using the LaTeX macro package with LLNCS style